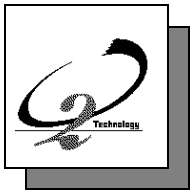


C Interface to O_2
User Manual

Release 5.0 - May 1998

Information in this document is subject to change without notice and should not be construed as a commitment by O_2 Technology.

The software described in this document is delivered under a license or nondisclosure agreement.

The software can only be used or copied in accordance with the terms of the agreement. It is against the law to copy this software to magnetic tape, disk, or any other medium for any purpose other than the purchaser's own use.

Copyright 1992-1998 O_2 Technology.

All rights reserved. No part of this publication can be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopy without prior written permission of O_2 Technology.

O_2, O_2Engine API, O_2C, O_2DBAccess, O_2Engine, O_2Graph, O_2Kit, O_2Look, O_2Store, O_2Tools, and O_2Web are registered trademarks of O_2 Technology.

SQL and AIX are registered trademarks of International Business Machines Corporation.

Sun, SunOS, and SOLARIS are registered trademarks of Sun Microsystems, Inc.

X Window System is a registered trademark of the Massachusetts Institute of Technology.

Unix is a registered trademark of Unix System Laboratories, Inc.

HPUX is a registered trademark of Hewlett-Packard Company.

BOSX is a registered trademark of Bull S.A.

IRIX is a registered trademark of Siemens Nixdorf, A.G.

NeXTStep is a registered trademark of the NeXT Computer, Inc.

Purify, Quantify are registered trademarks of Pure Software Inc.

Windows is a registered trademark of Microsoft Corporation.

All other company or product names quoted are trademarks or registered trademarks of their respective trademark holders.

Who should read this manual

This manual presents the basic concepts of the C interface to O_2 and describes how to link external C applications to O_2. It details the management of object handles, iterator functions, and how to use O_2Look and OQL with C. The interface with O_2C is also described, including method and function calls and object handle conversions to O_2C variables. A complete programming example is presented.

Other documents available are outlined, click below.

See O2 Documentation set.

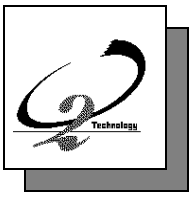


TABLE OF CONTENTS

This manual is divided into the following chapters:

- 1 - Introduction
- 2 - Runtime Library
- 3 - Example C application

TABLE OF CONTENTS

1 Introduction 9

1.1 System overview 10
Integrated modules 12
1.2 Features and advantages 13
1.3 Introducing the C interface to O2 14
C Application structure 15
1.4 Manual overview 17

2 Runtime Library 19

2.1 Database Connection 20
Starting an O2 session 20
Setting a base 22
2.2 Transaction Management 23
Defining transactions 23
Read-only transactions 23
Update transactions 24
2.3 Using OQL in C 26
o2query 26
Method and function call 28
2.4 Handle management 31
Manipulating functions 31
Iterator functions 33
Handle conversion 34
2.5 Running an application 37
Example 37
2.6 Garbage Collection 38
Example 38
2.7 Leaving the database 39

TABLE OF CONTENTS

3 An Example C Application 41

3.1 Introduction 42

3.2 Define the class Person in file Person.h 43

The class Person 44
Writing and reading a person from C 44
Implementing O2C read and write methods 45
Creating O2 objects from C 47
The Person_new C function 47
Implementing Person_new in O2C 47

3.3 The application code 49

The C header file Person.h 49
The O2C schema 49

3.4 Class Implementation in Person.c 52

3.5 The application in the main.c file 55

Main program 56

3.6 Compiling your application 58

The Makefile 58
Makefiles and Platforms 58

INDEX 61

TABLE OF CONTENTS

1 Introduction

GENERAL OVERVIEW OF THE
C INTERFACE TO O_2

Congratulations! You are now a user of the C interface to O_2!

You will find that developing O_2 applications in C has several key advantages: data persistence, conformance to C standards, reuse of existing C applications, extended C programming and openness to all O_2 tools including the OQL query language and O_2Look graphical interface generator.

This chapter introduces the O_2 interface to C.

The chapter is divided into the following sections:

- System overview
- Features and advantages
- Introducing the C interface to O2
- Manual overview

1 Introduction

1.1 System overview

The system architecture of O_2 is illustrated in Figure 1.1.

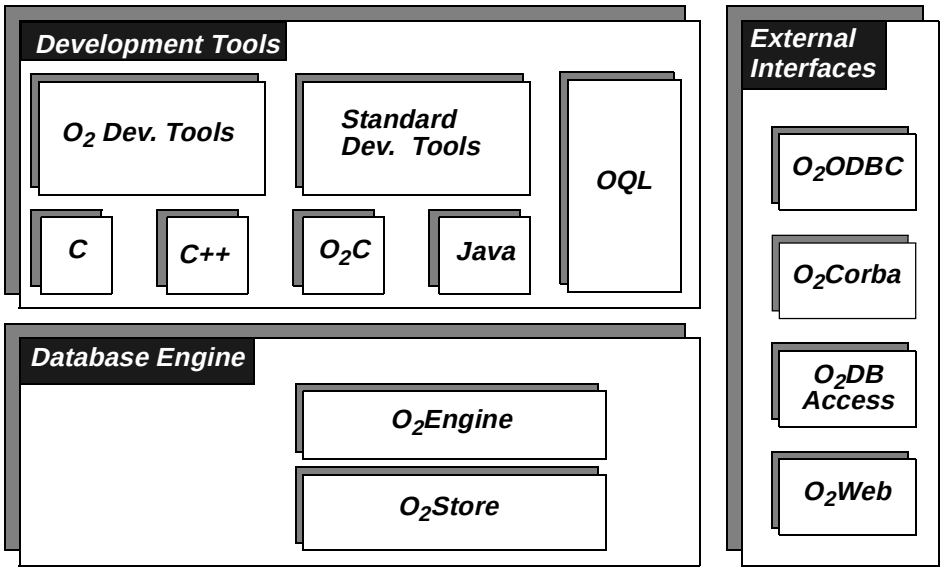


Figure 1.1: O_2 System Architecture

The O_2 system can be viewed as consisting of three components. The *Database Engine* provides all the features of a Database system and an object-oriented system. This engine is accessed with *Development Tools*, such as various programming languages, O_2 development tools and any standard development tool. Numerous *External Interfaces* are provided. All encompassing, O_2 is a versatile, portable, distributed, high-performance dynamic object-oriented database system.

Database Engine:

- O_2Store — The database management system provides low level facilities, through O_2Store API, to access and manage a database: disk volumes, files, records, indices and transactions.
- O_2Engine — The object database engine provides direct control of schemas, classes, objects and transactions, through O_2Engine API. It provides full text indexing and search capabilities with O_2Search and spatial indexing and retrieval capabilities with O_2Spatial. It includes a Notification manager for informing other clients connected to the same O_2 server that an event has occurred, a Version manager for handling multiple object versions and a Replication API for synchronizing multiple copies of an O2 system.

Programming Languages:

O_2 objects may be created and managed using the following programming languages, utilizing all the features available with O_2 (persistence, collection management, transaction management, OQL queries, etc.)

- C — O_2 functions can be invoked by C programs.
- C++ — ODMG compliant C++ binding.
- Java — ODMG compliant Java binding.
- O_2C — A powerful and elegant object-oriented fourth generation language specialized for easy development of object database applications.
- OQL — ODMG standard, easy-to-use SQL-like object query language with special features for dealing with complex O_2 objects and methods.

O_2 Development Tools:

- O_2Graph — Create, modify and edit any type of object graph.
- O_2Look — Design and develop graphical user interfaces, provides interactive manipulation of complex and multimedia objects.
- O_2Kit — Library of predefined classes and methods for faster development of user applications.
- O_2Tools — Complete graphical programming environment to design and develop O_2 database applications.

Standard Development Tools:

All standard programming languages can be used with standard environments (e.g. Visual C++, Sun Sparcworks).

External Interfaces:

- O_2Corba — Create an O_2/ Orbix server to access an O_2 database with CORBA.
- O_2DBAccess — Connect O_2 applications to relational databases on remote hosts and invoke SQL statements.
- O_2ODBC — Connect remote ODBC client applications to O_2 databases.
- O_2Web — Create an O_2 World Wide Web server to access an O_2 database through the internet network.

1 Introduction

Integrated modules

If you choose to use the C interface to O_2, you see that C inherits all the advantages of the O_2 system database engine O_2Engine:

- Objects in C applications are manipulated in the same way as standard C objects.
- Any existing C application can be made persistent after very minor modification.
- O_2 ensures database consistency, referential integrity and automatically manages the dynamic memory of O_2 objects. It also provides facilities for creating and manipulating O_2 and C collections.
- You use the complementary tools OQL and O_2Look: database query, complex value creation and initialization, object browsing and object editing.

1.2 Features and advantages

Developing O_2 applications in C has several key benefits: data persistence, conformance to C standards, reuse of existing C applications, extended C programming and openness to all O_2 tools including the OQL query language and O_2Look graphical interface generator.

C inherits all the advantages of the O_2 system database engine O_2Engine.

- Persistence

  The data in C applications can be made persistent with functions for creating, reading and writing objects. Your C applications can be made to have persistent structs after very minor modification. Persistent roots are defined dynamically in C.

  Persistence of data is achieved through the declaration of persistent roots to the application. Edit and display facilities for C data are generated automatically.

- Code reuse

  Existing C applications are easily extended and their objects made persistent. Code reuse by importing classes from different schemas.• Database consistency is maintained

- Improved and extended C

  O_2 has libraries for creating and manipulating O_2 and C collections. Generic libraries for easily creating and manipulating O_2 collections with reuse of existing libraries without recompilation OQL and O_2Look can be integrated and applied to C objects. Primitives for database transaction management are available from C applications

  Programming O_2 in C features include User-programmed transaction commit and abort. Standard C compiler. A C programmer needs no previous knowledge of O_2 to start developing applications immediately.

- Integrated tools

  You can take advantage of complementary tools: database query, complex value creation and initialization, object browsing and object editing. Complementary tools can be integrated with the C interface, such as database query, complex value creation and initialization as well as object browsing and editing from C.

1.3 Introducing the C interface to O_2

As O_2C is a superset of C, you will find that the object-oriented 4th generation language O_2C in itself has all the features you need.

However, you cannot use the O_2C application model if, for example, your application needs a main program written in plain C.

This manual explains how to write C programs which can use the O_2 database in order to store and retrieve C data. A a C programmer, you will find programming an application using the O_2 database very easy.

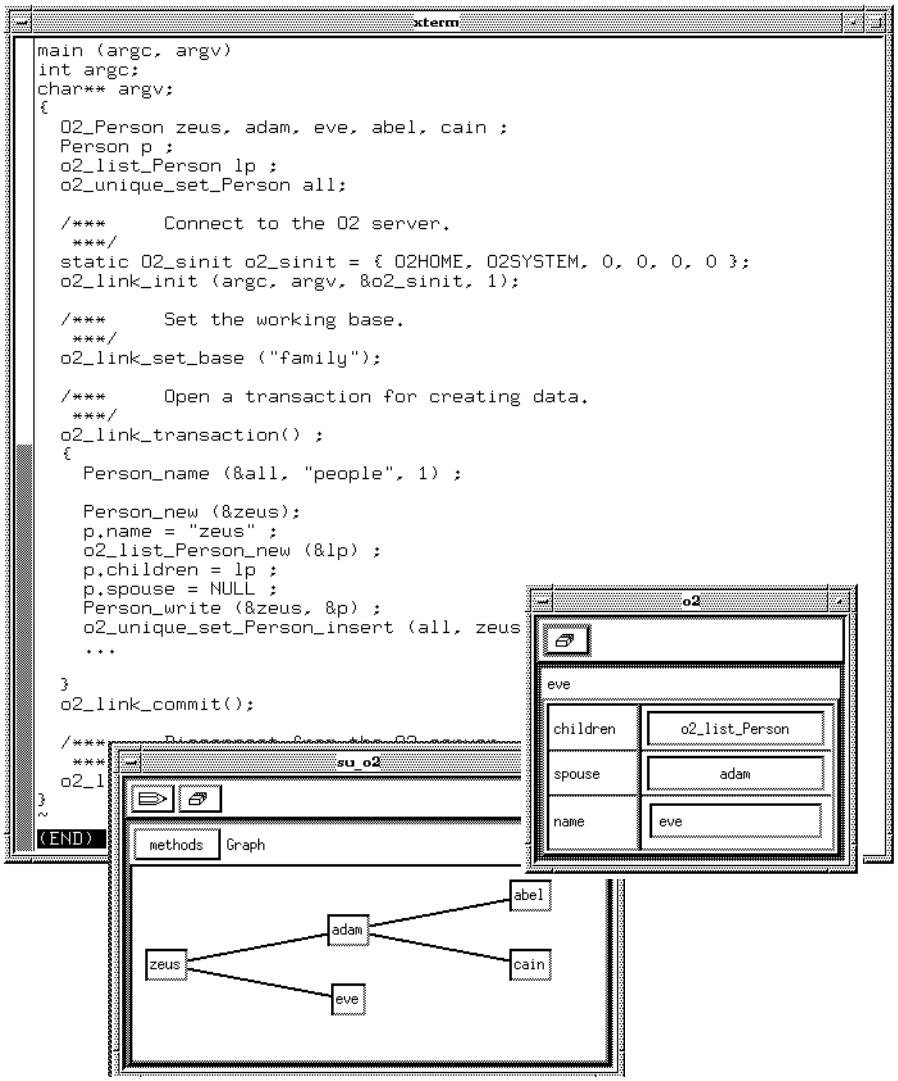


Figure 1.2: The C interface to O_2

C Application structure

An application is a C executable which at some moment connects itself to O_2, opens a database and then runs under the control of the O_2 transaction manager.

It then uses OQL to query the database and to call up O_2C functions and methods in order to manipulate handles (O_2 objects or values).

- Database connection

  A C application communicates with O_2 via various functions defined in an O_2 library.

  The C application starts up an O_2 session using the **o2_link_init** function and defines a database context against which the application runs using the **o2_link_set_base** function.

- Transactions

  It can then start up a transaction using the **o2_link_transaction** function, and can end the transaction by committing the transaction using the **o2_link_commit** function, validating the transaction using the **o2_link_validate** function or aborting the transaction using the **o2_link_abort** function.

  An application is usually made up of many transactions. each one reflecting a particular utilization of the database: creating objects, querying information, updating objects, adding new information, deleting information, etc.

  By default, an application runs in a read-only transaction. This is a transaction where no updates on persistent data are allowed.

- Using OQL in C

  Inside a transaction, the object query language OQL is used so that your C application can use an O_2 database in order to:

  - query an O_2 collection.
  - call an O_2C function
  - call an O_2C method

- Handles

  For a C application, an O_2 object or an O_2 value is a handle.

  Your C program manipulates these O_2 objects and values by calling O_2C functions or methods, and passing and receiving the objects and values as parameters.

You can carry out the following operations on handles:

- assign a value to another value
- assign an object to another object
- assign a value to a named value
- assign an object to a named object
- iterate on a collection.

You can also carry out handle conversion. Inside an O_2C function body or method body, you can convert a handle into an **o2** variable or conversely, convert an **o2** variable into a handle. You do this by using the O_2C function **c_assign**.

- Running the application

  You can start an O_2C application from a C program using the function **o2_run_application**.

- Leaving an O_2 session

  You end the session and leave the database using the function **o2_link_end**.

1.4 Manual overview

This manual is divided up into the following chapters:

- ***Chapter 1 - Introduction***

  This chapter introduces the C interface to O_2, outlines the O_2 system and gives all the features and advantages the interface has to offer. It then goes on to introduce the structure of a C application in O_2outlining such concepts as database connection, transaction management, using OQL to query the database and to call O_2 functions and methods in order to manipulate O_2 objects and values, handles (O_2 objects and values), running the application and leaving an O_2 session.

- ***Chapter 2 - Runtime Library***

  This chapter gives all the runtime library functions you need to use the C interface to O_2 and is divided up into the following sections:

  - Database connection
  - Transaction management
  - Using OQL in C
  - Handle management
  - Running the application
  - Leaving the database

- ***Chapter 3 - Example application***

  This chapter gives a complete example of how to use the C interface to O_2, showing how to make structures persistent, how to create data as well as store and retrieve these structures to and from a database.

1 Introduction

2 Runtime Library

This chapter gives the runtime library functions you need to use the C inter face to O_2.

The chapter is divided into the following sections:

- Database Connection
- Transaction Management
- Using OQL in C
- Handle management
- Running an application
- Garbage Collection
- Leaving the database

Note

Please refer to the O_2C and OQL manuals for more details on how to use these languages.

2 Runtime Library

2.1 Database Connection

To build the final O_2/ C application, the C interface has a runtime library through which the connection between O_2 and C is carried out.

Your C program communicates with O_2 by invoking the functions defined in the following libraries:

- **libo2lklink.a** and **libo2linkcuse.a** to use O_2Look inside O_2C methods and functions,
- **libo2link.a** and **libo2linkcuse.a** - these do not include O_2Look.

The external application must be linked to the O_2 library, situated in the **lib** subdirectory of the O_2 installation directory.

If the application uses O_2Look, it must be linked to MOTIF (1.2) and standard X libraries.

This section is divided up into the following sections:

- Starting an O2 session
- Setting a base

Starting an O_2 session

To use O_2 in a C program, the application program must start an O_2 session which initializes the connection to the server.

To do this your application uses the O_2 library function:

```
int o2_link_init (argc, argv[], sinit, mode)
int argc; char *argv[]; O2_sinit *sinit; int mode;
```

The **o2_link_init** function sets up a working connection between the external C program and the specified O_2 system name and is much like starting up an O_2 session under Unix, by invoking the **o2** executable program.

The function passes the **argc** and **argv** parameters as command line arguments to the **o2** executable. The **argc** and **argv** parameters are the number and list of command line arguments with which the external C application was launched.

Any valid O_2 command line options can be used when launching a C application.

The **sinit** parameter is a pointer to an instance of the following structure:

```
typedef struct {
 char *sysdir;
 char *sysname;
 char *svname;
 char *swapdir;
 char **libpath;
 char **libname;

} O2_sinit;
```

An **O_2** installation directory
An **O_2** named system
Machine where **o2server** is running
Special swap directory
Paths to search for libraries
Library names to search for

The definition of this structure is given in the header file **o2_c.h**.

All the various members of the structure are described below.

The first four members of the structure identify which O_2 system to use, and the last two only serving those applications needing dynamic run-time linking to one or more libraries of C functions:

- **sysdir:**

  This is the absolute path of an O_2 installation directory, containing the **systems** file and the **bin** subdirectory holding the O_2 executables. This must not be null.

- **sysname:**

  The name of the O_2 database system working with the C application. This must be defined in the **systems** file indicated by **sysdir** and must not be null.

- **svname:**

  The name of a machine on the local network on which the **o2server** process is running. If null, **o2_link_init** looks for the server specification in the **systems** file in the O_2 installation directory.

- **swapdir:**

  The absolute path of a directory for special swap files. If null, the default location (O_2 installation directory **swap** subdirectory) is used.

- **libpath:**

A null terminated array of absolute paths which are searched, for libraries during a dynamic loading procedure for C functions. This is similar to the **-L** option of the **o2** executable. It is important to C programs that invoke dynamically-linked modules from user-specified libraries; otherwise this can be null.

- **libname:**

  A null terminated array of library names which are each searched, for dynamically-loaded functions called by the application. This is similar to the **-l** option of the **o2** executable. It is important to C programs that invoke dynamically-linked modules from user-specified libraries; otherwise this can be null.

Finally, the **mode** parameter[1] is an integer which determines whether or not the external application uses the O_2Look graphic user interface. Zero indicates that the application uses O_2Look; 1 indicates that the application is alphanumeric.

The **o2_link_init** function returns zero if the link to O_2 was successfully established or an error code if not.

Setting a base

The C program then sets the database with which it wants to work by using the following O_2 function:

```
int o2_link_set_base (base_name) char *base_name;
```

This is the same as the O_2 schema command set base.

The **o2_link_set_base** function identifies the O_2 base with which the external program wants to work.

A base with the specified name must already exist within the named database system specified in the **o2_link_init** function above.

The function returns zero if the base indicated is found or an error code if not.

1. Only meaningful with the graphical version of O2

2.2 Transaction Management

This section describes all the various runtime library functions you need to work with transactions.

This section is divided as follows:

- Defining transactions
- Read-only transactions
- Update transactions

Defining transactions

You start up a new transaction using the following function:

```
void o2_link_transaction()
```

By default, a C application runs as a program, i.e. it is allowed to read persistent data but not to update it. In order to update, the application must start a transaction.

Read-only transactions

The call to the **o2_link_transaction** is not obligatory is you only want to read data because by default, an application runs in a read-only transaction.

This is a transaction where no updates on persistent data are allowed and no locks are set by the system.

After a transaction ends, the application is automatically reset into a read-only transaction. Such a read-only transaction is useful because it does not conflict with other transactions.

It can read any objects or collections, and it can create any temporary O_2 objects or collections as complex as needed.

The read-only transaction is therefore ideally suited to that part of your application where an interactive dialogue with the end-user occurs, or where some complex calculation is necessary after reading stable data.

This part of an application is usually followed by a standard transaction which manages the creation and updating of persistent data.

Note

When you start a transaction using **o2_link_transaction**, O_2 ensures that the objects you access are always up-to-date, i.e. with the latest committed value.

Moreover, the transaction protects you from concurrent updates and guarantees that the data you read will not change during the transaction. Inside the transaction, you simply must check that the assumptions made in the read-only phase, if any, still hand.

Update transactions

- ***Link***

  An update transaction begins with the function:

  ```
  void o2_link_transaction()
  ```

  This leaves read-only mode. Consistency is guaranteed by O_2 which automatically rereads the variables which have been read during read-only mode.

- ***Commit***

  The following transaction ends the current transaction on the current database:

  ```
  void o2_link_commit()
  ```

  o2_link_commit updates the database, releases all locks, frees all memory allocated at runtime and then starts a read-only transaction. All changes made to the database during the present transaction are written to disk and a new transaction is initiated. After a commit, all handles held by a C application are no longer valid and they must be recomputed starting from persistent roots.

- ***Validate***

  The validate function is as follows:

  ```
  void o2_link_validate()
  ```

This function is the same as **o2_link_commit** except that it does not free the memory allocated by O_2 to run this transaction. After a validate, all handles held by a C application are still valid.

- ***Abort***

  The abort function is as follows:

```
void o2_link_abort()
```

  This function also ends the current database transaction. Any changes made to the database during the present transaction are abandoned and a new transaction is initiated. After an abort, all handles held by a C application are no longer valid and they must be recomputed starting from persistent roots.

2.3 Using OQL in C

This section now describes how to call OQL from a C program using this function, pass C parameters to it and get the result of the query.

It then describes how to call an O_2 method or function from OQL in order to use the O_2 database.

o2query

The C interface to O_2 has a set of functions, each with a name beginning with **o2query** which are the interface between C applications and OQL.

The first argument is a pointer to the result returned by the query.

The function returns a zero unless there is a syntax error in the query specified in *querytext*.

The name and synopsis of the function depends on the type of the expected result:

- ***object or value***

  If the query result is an object or value (tuple, (unique) set, list).

  ```
  int o2query (void** handle_address,
               char *querytext [, qvariable]...)
  ```

- ***integer***

  If the query result is an integer:

  ```
  int o2query_integer (int *i,
                       char *querytext [, qvariable]...)
  ```

- ***real***

  If the query result is a real number:

  ```
  int o2query_real (double *d,
                    char *querytext [, qvariable]...)
  ```

- ***character or boolean***

  If the query result is a character or boolean:

  ```
  int o2query_char (char *c,
                    char *querytext [, qvariable]...)
  ```

- ***string***

  If the result of the query is a string:

  ```
  int o2query_string (char **s,
                      char *querytext [, qvariable]...)
  ```

  o2query_string dynamically allocates a C array for the query result and it is you, the user who frees this **char*** when it is no longer used.

  The other two arguments of the **o2query** function are:

- ***querytext***

  A string, containing the text of the query, and can include references to O_2 schema names but not to local program variables.

  The string can, however, contain references to query variables (defined by subsequent *q_variable* parameters, described below), in the form of **$1** for the first variable, **$2** for the second, etc.; and these query variables refer to local program variables. See the examples below.

- ***q_variable***

  An expression representing a value to be substituted for the **$n** references in the *querytext* argument.

  There must be as many of these *q_variable* arguments as there are distinct **$n** references.

  Each query variable identifies the type as well as the value of a **$n** reference and must therefore take one of the following forms:

  - **o2_integer (int)**, for a C integer variable
  - **o2_real (double)**, for a C floating-point number variable
  - **o2_char (char)**, for a C character variable
  - **o2_string (char *)**, for a C string variable
  - **o2_handle (**x**)**, where x is a handle standing for an O_2 object or an O_2 complex value (tuple, list, set, unique set)

- ***Example queries***

  To find the value of the **age** attribute of named object **Father**:

  ```
  int i;
  o2query_integer (&i, "Father.age");
  ```

To filter the named list value **Employees** and select only those employees between the ages of **min** and **max**:

```
typedef void* o2_list_Person;
o2_list_Person s = 0;
int min, max;
...
o2query (&s, "select e
     from e in Employees
     where e.age < $2 and e.age > $1",
           o2_integer(max), o2_integer(min));
```

To test if Mister M. belongs to this set of employees:

```
char is_here;
char M_name[10];
...
o2query_char (&is_here, "exists p in $1: p.name
= $2",
              o2_handle(s), o2_string(M_name));
```

The **o2query** function is also used to initialize C variables which are instances of collections.

This is probably the best way to initialize such variables.

You can write for instance:

```
typedef void* o2_list_integer;
o2_list_integer L = 0;
o2query (&L, "list (1, 4, 9, 16, 25)");
```

Method and function call

An O_2C method or function is easily called using OQL.

For example, to call the method **display** on a handle **h** and then call the function **stats**, you use the following OQL queries:

```
void* Void = 0;
int rc;
double max;

o2query(&Void,
      "$1->display",
      o2_handle(h));

o2query_integer(&rc,
           "stats($1, $2)",
      o2_handle(h),
      o2_real(max));
```

From O_2C, when a method or function is called, it receives handle parameters as normal O_2C parameters, provided their types match the corresponding O_2C parameters.

For example, the O_2C function **stats** is declared as:

```
class Vector type list(real) end;

function stats(v: Vector, upper_bound: real): integer;
```

At calling time, if the handle **h** is not an object of class **Vector** (or one of its subclasses) a **type error** exception is raised.

Likewise an object or a value returned by an O_2C method or function is received as a handle by C.

For example, you have a C handle representing a list of integers and you want to call an O_2C function which adds an integer to this list and returns the new inserted list.

```
function insert (l: list(integer),
                 e: integer): list(integer);

function body insert{
   l += list(e);
   return l;
}
```

You call the function from C as follows:

```
void * vector;
   ...
   o2query(&vector,
           "insert($1, 100)",
           o2_handle(vector));
```

2.4 Handle management

A handle represents an O_2 object or an O_2 value inside an application written in C.

Manipulating functions

The following C functions allow you to manipulate handles. To use these functions, a C program must include the file header: **o2_c.h**.

The **o2_assign_handle** function assigns the source handle to the target handle:

```
void o2_assign_handle (target_handle_address,
source_handle)
void** target_handle_address; void* source_handle}
```

Before assignment, it automatically de-references the handle contained in the target variable, if one exists, and consequently, at the time of call, the target handle must be set to 0 or else must contain a valid handle.

Moreover, if the target handle is not 0 before assignment, it must contain a handle of the same kind as the one contained in the source handle, i.e. if source contains a value, target must be a value and if source is an object, target must also be an object.

Note that a handle is freed explicitly and set to 0 by the **o2_unref_handle** function below.

If the above conditions are not satisfied, the assignment fails and the application crashes possibly with the message “Internal inconsistency in the Complex Object Manager” followed by an automatic abort.

For example, to get a handle of a named collection **people** into a variable called people and assign it to another variable, called **copy_people**.

```
#include "o2_c.h"
typedef void* o2_set_Person;
 o2_set_Person people = 0, copy_people = 0;
 o2query(&people, "people");
 o2_assign_handle(&copy_people, people);
```

The function:

```
void o2_assign_name (name, source_handle) char* name;
void* source_handle
```

updates the named object (or value) identified by **name** with the object (or value) found in **source_handle**. The name must be defined in the current base.

For example

```
void* node = 0;
compute_root(&node);
o2_assign_name("Graph_root", node);
```

The **o2_unref_handle** is a macro which frees the handle passed as parameter and sets this handle to 0:

```
void o2_unref_handle (handle)
```

The parameter must contain a valid handle or else 0. Otherwise the application would crash somewhere.

This macro must be called as soon as a handle variable is no longer used by the application, and allows the O_2 system to release and garbage objects and values which are no longer reachable.

Keeping references to unused objects or values can slow performance.

For example

```
   void* temp = 0;
  o2query(&temp, "people"); /* temp stands for a set value */
   ...
 /* to reuse temp and assign to it a new value of a new
    type (list(string)), you must free it before assignment */
   o2_unref_handle(temp);
   o2query(&temp, "list('blue', 'white', 'red')");
   o2query(&temp, "list("'yellow')"); /* same type */

/* cannot assign temp with an OBJECT without freeing it
      before: */
   o2_unref_handle(temp);
   o2_assign_handle(&temp, node);
```

Iterator functions

These functions enable you to iterate on a collection (object or value) and use a type definition **o2_iterator**.

The function **o2_iterator_open** initializes a scan passed as parameter:

```
o2_iterator o2_iterator_open (collection_handle)
```

This function returns **o2_iterator** which must be passed to the **next** and **close** macros below.

The function:

```
int o2_iterator_next (iterator, element_address)
int o2_iterator_next_string (iterator,
char_array_address)
```

gets the next element from the collection for which the iterator has been opened. If the scan is exhausted, it returns 0, or returns 1 if not.

The next element is the address passed as second parameter and must be the address of:

- a **char**, for collections of booleans or chars
- an **int**, for collections of integers
- a **double**, for collections of reals
- a **handle**, for collections of complex values or collections of objects. The handle is set to 0 before or is valid and consistent, as with **o2_assign_handle**.

The function **o2_iterator_next_string** gets the next element from a collection of O_2 strings or bits and puts the string into a C array.

This array is defined by the second parameter which is of type **char *** and, which is allocated by the application before calling this function and must be big enough to hold the string.

The function **o2_iterator_close** terminates the scan:

```
o2_iterator_close (iterator)
```

For example for collection of objects:

```
o2_set_Person s = 0;
o2_Person p = 0;
o2_iterator it;
Person val;
o2query(&s, "people");
it = o2_iterator_open(s);

while(o2_iterator_next(it, &p)){
  Person_read(&val, p);
  print(val);
}
o2_unref_handle(p); /* release reference "p" to the
                       last collection element */
o2_iterator_close(it);
```

For collection of strings:

```
char element[20];
void* flag = 0;
o2_iterator it;
o2query(&flag, "list('blue', 'white', 'red')");

it = o2_iterator_open(flag);
while(o2_iterator_next_string(it, element)){
   printf("%s\n", element);
}
o2_iterator_close(it);
```

Handle conversion

Sometimes you have to convert a C handle into an O_2C variable or conversely, put O_2C data into C types.

For example, a C program wants to pass a C struct or a C array, containing handles, to an O_2C function (or method).

To interpret these handles, O_2C transforms it to normal O_2C typed data.

Conversely, an O_2C function (or method) may need to put O_2C data into a C struct or a C array to deliver it to a calling C program. **c_assign** carries out these transformations:

- ***handle to variable***

  To convert a handle into an O_2C variable, you use the following function:

  c_assign(*O2c_variable, handle***)**

  This O_2C function assigns the handle *handle*, an integer in O_2C, into the O_2C variable *O2c_variable*.

  This variable is an object or a complex value of a type consistent with the actual data referred to by the handle. Dynamic type checking occurs (except when compiled with the O_2C option -O).

- ***variable to handle***

  To convert a variable into a handle, you use the following function:

  c_assign(*handle_variable, O2C_expression***)**

  This O_2C function assigns the O_2C expression into a C handle, an integer in O_2C. The expression delivered an object or a complex value which allows handles to be returned to a C program.

  For example, you can write an O_2C function which gets from a C program a 3-field C structure: a handle representing an object of type Employee, a double representing the new salary for this employee and a handle for an object describing the employee category.

  The O_2C function updates the employee's salary, computes the new category from the new salary and returns it in the C structure.

  The structure address is passed as an O_2C integer and the structure type is described in the C file: **employee.h**:

```
typedef void* handle;
typedef struct{ handle employee;
                double salary;
                handle category;} employee;
```

The O_2C function is:

```
type employee_address: integer;
function promote(this: employee_address);

function body promote{
#include "employee.h"
   o2 Employee emp;
   o2 Category cat;
   employee* e= (employee*) this;
   c_assign(emp, e->employee); /* get the Employee: */
   emp->salary = e->salary; /* update his salary */
   cat = compute_cat(e->salary); /* compute new category */
   c_assign(e->category, cat); /* return it */
}
```

The C program calls the O_2C promote function as:

```
#include employee.h
void* Void = 0;
employee smith;
smith.employee = 0; /* Careful with handle assignment */
smith.category = 0;
o2query(& smith.employee, "Smith");
smith.salary = 15000.0;
o2query(&Void, "promote($1)", o2_integer(&smith));
```

2.5 Running an application

You can start an O_2C application from a C program by using the following function:

```
int o2_run_application (char * application_name,
      char * parameters)
```

o2_run_application has the following parameters:

- ***application_name***

  This is the name of the O_2C application.

- ***parameters***

  The *parameters* option gives the parameters with the syntax **(param1, param2...)**. This option equals NULL if the application has no parameters.

Important

Before using **o2_run_application**, you must start up a transaction.

This function returns 1 if the application is unknown, otherwise 0.

Example

```
o2_run_application  ("customers",
      "(1994, \"September\")";
```

2.6 Garbage Collection

To clean up the base, you can use the following function:

```
void o2_garbage_base(char *base_name)
```

The **o2_garbage_base** function collects the unreachable objects of a given base. To use this function you must call it explicitly from time to time.

To prevent any temporary references onto persistent data, you must commit before calling the garbage collector. If you do not do this you may get wild references.

Note

As garbage collection works within a base, this means that those objects in a particular base that are only reachable from another base, will also get garbage collected .

Example

```
o2_link_commit();

o2_link_transaction();

o2_garbage_base("my_base");
```

2.7 Leaving the database

To end an O_2 session

```
void o2_link_end()
```

The **o2_link_end** function terminates the current O_2 session. **o2_link_commit** is automatically carried out and the connection established by **o2_link_init** is then broken.

The C application can then, of course, continue to perform tasks unrelated to O_2.

2 Runtime Library

3 An Example C Application

This chapter aims to illustrate how to use the C interface to O_2 with a simple example that outlines all the basic steps to writing C programs with O_2.

This chapter is divided up into the following sections:

- Introduction
- Define the class Person in file Person.h
- Class Implementation in Person.c
- The application in the main.c file
- Compiling your application

3 An Example C Application

3.1 Introduction

To program C using the C interface to O_2, you must go through the following steps

- Define the C classes, the instances of which you want to make persistent. In this tutorial this is the class **Person** in **Person.h**.
- Define the implementation of the class **Person** in **Person.c**
- Write your application
- Compile your application and link it to the relevant O_2 library.

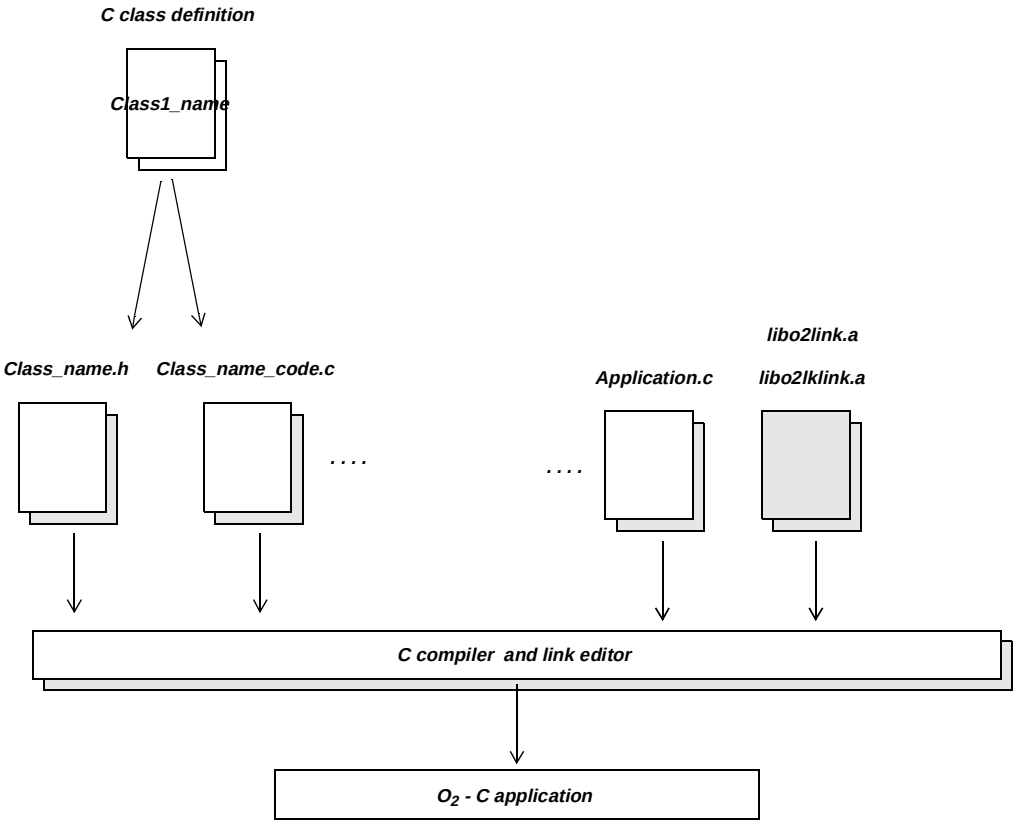


Figure 3.1: C Interface to O_2

3.2 Define the class Person in file Person.h

First of all, you must make your C structures persistent. As A pointer is only valid inside memory, you must change the type definition and replacing all the pointer declarations with handles.

For example

```
typedef struct person{
   char *  name;
   int age;
  person[]* children;  /* dynamic  array of persons */
   person* spouse;
   struct {
     int number;
     char * *  forename;
   } forenames;
} person;
```

Recall that a handle is a **void*** and it is useful to give different symbolic names to handles in order to document the O_2 data storing the information.

For example, you define a **person** as an O_2 object and so the field **spouse** is also an object. You decide to to put **children** into a list of persons. For future operations on person objects, it is useful to complete the structure by a transient (i.e. non persistent) attribute containing the object identifier.

In fact a handle contains a unique identifier for the object it represents and an application preserves object identity.

This is not so easy a task and requires sophisticated data structures, e.g. a hash table which maps the handle value into the memory address of the corresponding C structure.

Finally, in the same file, you make provision for future type extensions like **o2_set_Person**, or function declarations like **Person_new()** which returns a newly created **person**, the definition of which is given later.

The persistent C type therefore looks as follows:

3 An Example C Application

```
/* file "Person.h"   is: */
typedef void* o2_Person;
typedef void* o2_list_Person;
typedef void* o2_set_Person;
typedef void* o2_unique_set_Person;
typedef struct{
   char *  name;
   int age;
   o2_list_Person children;
   o2_Person spouse;
   struct {
     int number;
     char * *  forename;
   } forenames;
   o2_Person object_id;
} person;
extern o2_Person Person_new();  /* defined further */
```

The class Person

You now know how to represent a “person” in O_2.

So now define the class with the corresponding O_2 type.

```
class Person type tuple(
   name: string,
   age: integer,
   children: list(Person),
   spouse: Person,
   forenames: list(string))
end
```

Writing and reading a person from C

You now have a C type and an O_2 class and you are ready to define the two C functions:

Person_read and **Person_write**.

The first takes an O_2 object and puts it in a C struct, while **Person_write** copies a C struct into an O_2 object.

You cannot directly write these conversions in C but in O_2C. So the two C functions simply call two corresponding O_2C methods of the class Person: write and read:

```
void Person_read(val, p)
   person* val; o2_Person p;
{
   /* reads the value of the Person p into the C struct */

   void* Void = 0;
   o2query(&Void, "$1->read($2)", o2_handle(p),
o2_integer(val));
   val->object_id = p;

/* Note the object id inside the structure
   This is useful to connect this value to the object. */

};
void Person_write(p, val)
   o2_Person p; person val;
{
   void* Void = 0;

   /* updates the value of the Person p from the C struct */

   o2query(&Void, "$1->write($2)", o2_handle(p),
o2_integer(&val));
};
```

Implementing O_2C read and write methods

- ***Read method***

  The read method takes a pointer to the C struct and copies the object into it.

```
type pointer: integer;
method body read(val: pointer) in class Person{
#include "Person.h"
#include <malloc.h>
   person* this = (person *) val;
   this->name = malloc(count(self->name)+1);
   strcpy(this->name, self->name);
   this->age = self->age;
   this->children = 0; this->spouse = 0;
   c_assign(this->children, self->children);
   c_assign(this->spouse, self->spouse);
   { o2 string fn; int i = 0;
     this->forenames.number = count(self->forenames);
     this->forenames.forename = malloc(sizeof(char*) *
           this->forenames.number);
     for(fn in self->forenames){
        this->forenames.forename[i] = malloc(count(fn)+1);
        strcpy(this->forenames.forename[i], fn);
        i++;
     }
   }
};
```

- ***Write method***

  The write method takes a pointer to the structure and copies the C struct into the object.

```
method body write(val: pointer) in class Person{
#include "Person.h"
   person* this = (person *) val;
   strcpy(self->name, this->name);
   self->age = this->age;
   c_assign(self->children, this->children);
   c_assign(self->spouse, this->spouse);
   { int i; self->forenames = list();
     for(i=0; i<this->forenames.number; i++)
        self->forenames += list((o2 string) this->forenames.forename[i]);
   }
};
```

Creating O_2 objects from C

To create an O_2 object in C, you define a C function which calls an O_2C function to create and initialize this object.

Notice how the O_2C function returns its parameter.

The type is a normal **Person** type, and the C function receives a handle.

The Person_new C function

```
o2_Person Person_new(name)
 char* name;
{
   /* creates a new persistent Person */
   o2_Person p = 0;
   o2query(&p, "Person_new($1)", o2_string(name));
   return p;
};
```

Implementing Person_new in O_2C

```
constant name people: set(Person);
/* the persistent extension */
method body init(name: string) in class Person{
/* constructor */
   self->name = name;
   people += set(self);
};
function Person_new(name:string): Person;
/* interface for C */
function body Person_new{
   return new Person(name);
};
```

Now you can write the complete application, enhancing it with fancy operations.

3 An Example C Application

The next section gives the code followed by the Makefile to generate the executable.

3.3 The application code

The C header file Person.h

```
typedef void* o2_Person;
typedef void* o2_list_Person;
typedef void* o2_set_Person;
typedef void* o2_unique_set_Person;
typedef struct{
   char *  name;
   int age;
   o2_list_Person children;
   o2_Person spouse;
   struct {
     int number;
     char * *  forename;
   } forenames;
   o2_Person object_id;
} person;
extern o2_Person Person_new();
```

The O_2C schema

```
schema family_s;
type pointer: integer;
class Person type tuple(
   name: string,
   age: integer,
   children: list(Person),
   spouse: Person,
   forenames: list(string))
method
   init(name:string),
   public birth(child: Person),
   public ancestors: unique set(Person),
   /* communication O2 <-> C methods: */
   public read(val: pointer),
   public write(val: pointer)
end;
```

3 An Example C Application

```
constant name people: set(Person);
/* persistent extension */
name God: Person;   /* a particular Person ! */
set class Person;
method body init(name: string){
   self->name = name;
   people += set(self);
};

method body birth(child: Person){     /* new child is born */
   self->children += list(child);
   if (self->spouse != nil)
      self->spouse->children += list(child);
};
method body ancestors: unique set(Person){
/* What are the ancestors of this Person ? */
   o2 unique set(Person) parents, ancestors;
   o2query(parents,
           "select unique x from x in people\
            where $1 in x.children",
            self);
   o2query(ancestors,
            "flatten(\
               select unique a->ancestors\
               from a in $1\
             )union $1"
            , parents);
   return ancestors;
};



method body read(val: pointer){
                          /* read a C val from this O2 object */
#include "Person.h"
#include <malloc.h>
   person* this = (person *) val;
   this->name = malloc(count(self->name)+1);
   strcpy(this->name, self->name);
   this->age = self->age;
   this->children = 0; this->spouse = 0;
   c_assign(this->children, self->children);
   c_assign(this->spouse, self->spouse);
   { o2 string fn; int i = 0;
     this->forenames.number = count(self->forenames);
     this->forenames.forename = malloc(sizeof(char*) *
this->forenames.number);
```

```
          for(fn in self->forenames){
              this->forenames.forename[i] = malloc(count(fn)+1);
           strcpy(this->forenames.forename[i], fn);
        }
      }
   };

   method body write(val: pointer){
   /* write a C val into this  O2 object */
   #include "Person.h"
      person* this = (person *) val;
      strcpy(self->name, this->name);
      self->age = this->age;
      c_assign(self->children, this->children);
      c_assign(self->spouse, this->spouse);
      { int i; self->forenames = list();
        for(i=0; i<this->forenames.number; i++)
           self->forenames += list((o2 string)
this->forenames.forename[i]);
      }
   };
   /* Special C interfaces  */
   function Person_new(name:string): Person;
   function body Person_new(name:string): Person{
      return new Person(name);
   };

   function clear;
   function body clear{
      people = set();
   };

   /*-------- Create a database ----------*/
   create base family;
   /*-------- ... and possibly index ----------*/
   create index people on name;
   create index people on children;
   quit
```

3.4 Class Implementation in Person.c

The following gives the code which implements the class **Person**.This is in the file **Person.c**.

```
#include "Person.h"
#include "o2_c.h""
#include <stdio.h>
#include <malloc.h>
#include <string.h>

o2_Person Person_new(name)
 char* name;
{
   o2_Person p = 0;
   o2query(&p, "Person_new($1)", o2_string(name));
   return p;
};
void Person_read(val, p)
   person* val; o2_Person p;
{
   void* Void = 0;
   o2query(&Void, "$1->read($2)", o2_handle(p), o2_integer(val));
   val->object_id = p;
};

void Person_write(p, val)
   o2_Person p; person val;
{
   void* Void = 0;

   o2query(&Void, "$1->write($2)", o2_handle(p), o2_integer(&val));
};
void Person_birth(p, child)
   o2_Person p; o2_Person child;
{
   void* Void = 0;
   o2query(&Void, "$1->birth($2)", o2_handle(p), o2_handle(child));
};
o2_unique_set_Person Person_ancestors(p)
   o2_Person p;
{
   o2_unique_set_Person s = 0;
   o2query(&s, "$1->ancestors()", o2_handle(p));
   return s;
};
void clear(){
   void* Void = 0;
   o2query(&Void, "clear()");
};
void person_init(val)
   person* val;
```

Header files

create a new persistent Person

read value of p into C struct

This value is of object id.

update value of p from C struct.

```
{
/* initialize a "person" structure: any "handle" must be set
   to 0, in order for future o2_assign_handle to work properly */
   val->name = 0;
   val->spouse = 0; val->children = 0;
   val->forenames.number = 0;
}
void person_clean(val)

   person* val;
{
 /* frees mallocated parts of a C struct */
   int i;
   free(val->name);  val->name = 0;
   for(i=0; i<val->forenames.number; i++)
      free(val->forenames.forename[i]);
   if(val->forenames.number >0){
       free(val->forenames); val->forenames.number = 0;
   }
   /* release internal handles: */
   o2_unref_handle(val->spouse);
   o2_unref_handle(val->children);
};

int has_forename(p, forename)
   person p; char* forename;
{
   int i;
   for(i = 0; i<p.forenames.number; i++)
      if(!strcmp(p.forenames.forename[i], forename)) return 1;
   return 0;
};

void add_forename(p, forename)
   person* p; char* forename;
{
   if(!has_forename(*p, forename)){
      p->forenames.forename = (char**)
                              realloc(p->forenames.forename,
                               sizeof(char*) * (++(p->forenames.number)));
      p->forenames.forename[p->forenames.number-1] = forename;
   }
};
```

3 An Example C Application

```
void print(val)
   person val;
{
   int i;
   printf("-- %s", val.name);
   printf(" ( %d years) ", val.age);
   for(i=0; i<val.forenames.number; i++)
      printf("%s", val.forenames.forename[i]);
   printf("\n");
};

/* Using Iterator */
void print_persons(s)
   o2_set_Person s;
{
   o2_Person p = 0;
   o2_iterator it;
   person val;
   person_init(&val);
   it = o2_iterator_open(s);
   while(o2_iterator_next(it, &p)){
     Person_read(&val, p);
     print(val);
     person_clean(val);   /*free memory allocated for reading object*/
   }
   o2_unref_handle(p);       /* release the object refered to by "p" */
   o2_iterator_close(it);
};

/* Keeping object references */

void mariage(val, with)
   person* val; o2_Person with;
{
   person the_spouse; person_init(&the_spouse);
   o2_assign_handle(&val->spouse, with);
  /* val->spouse gets a new reference to the spouse Person (with) ... */
   Person_read(&the_spouse, with);

   o2_assign_handle(&the_spouse.spouse, val->object_id);
    /* and the spouse refers to this Person. */
   Person_write(with, the_spouse);
   person_clean(the_spouse);
};
```

3.5 The application in the main.c file

The following code is the main program of our example application.

```
#include "o2_c.h"
#include "Person.h"

static void transaction_load(){         /* Load the database */
   o2_Person  zeus, adam, eve, tmp;
   person ad; person_init(&ad);
   o2_link_transaction(); /* Start a new update transaction */
   clear();  /* Create a new O2 object ... */
   zeus = Person_new("zeus"); /* and give him an O2  name. */
   o2_assign_name("God", zeus);
   adam = Person_new("adam");
   Person_birth(zeus, adam);
   eve  = Person_new("eve");
   Person_birth(zeus, eve);
   Person_read(&ad, adam);
   mariage(ad, eve);
   Person_write(adam, ad);
   Person_birth(adam, tmp = Person_new("abel"));
   o2_unref_handle(tmp);
   Person_birth(adam, tmp = Person_new("cain"));
   o2_unref_handle(tmp);
   person_clean(ad);
  o2_unref_handle(zeus); o2_unref_handle(adam); o2_unref_handle(eve);
   o2_link_commit();
}

static o2_Person cain; /* A global variable */
static void transaction_consult(){     /* Read-only transaction. */
   int rc;
   o2_set_Person people = 0;
   printf("----- ALL THE PEOPLE -------\n");
   o2query(&people, "people");
   print_persons(people);
   printf("----- CAIN'S ANCESTORS -------\n");
   cain = 0;
   rc =
    o2query(&cain,
          "element(select p from p in people where p.name = 'cain')");
   if(rc!=0){
      printf("query failed\n");
      o2_link_abort();
   }else{
      print_persons(Person_ancestors(cain));
   }
   o2_unref_handle(people); o2_unref_handle(cain);
}
```

3 An Example C Application

```
static void transaction_update(){
   o2_Person p, q;
   person jones; person_init(&jones);

  /* Start a new update transaction */

   o2_link_transaction();
   Person_birth(cain, p = Person_new("jones"));
   Person_read(&jones, p);
   add_forename(jones, "Charles");
   add_forename(jones, "Jean");
   add_forename(jones, "Christian");
   add_forename(jones, "Jean");
   jones.age = 20;
   Person_write(p, jones);
   Person_birth(p, q = Person_new("Smith"));
   print_persons(Person_ancestors(q));
   person_clean(jones);
   o2_unref_handle(p); o2_unref_handle(q);
   o2_link_commit();
}
```

Main program

The main program of the application in the file main.cc is as follows:

The application in the main.c file

```
main(argc, argv)
 int argc; char* argv[];
{
   static  O2_sinit o2_sinit = {
      "/o2_directory/home",
      "user_system",
      "machine",
      ".",
       0,0
   };
#define ALPHA 1
   if(o2_link_init(argc, argv, &o2_sinit, ALPHA)!=0){
     printf("Something wrong to start o2\n");
     exit(1);
   }
   o2_link_set_base("family");
   transaction_load();
   transaction_consult();
   transaction_update();
   {o2_set_Person people = 0;
     printf("----- ALL THE PEOPLE -------\n");
     o2query(&people, "people");
     print_persons(people);
   }
   o2_link_end();
}
```

3.6 Compiling your application

The Makefile

To compile this application, you can write the following Makefile (for HP).

```
LINK.c= cc $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)
O2HOME = /o2_directory/home
IFLAGS= -I. -I$(O2HOME) -I$(O2HOME)/include
CFLAGS = -g  $(IFLAGS)
LDLIBS = -Wl,-E  -L$(O2HOME)/lib -lo2link -lo2 -lo2linkcuse -lm  -ldld
#Using O2Look
# LDLIBS = -Wl,-E  -L$(O2HOME)/lib -lo2lklink -lo2 -lo2linkcuse -lm
-ldld -lXm -lXt -lX11
#Using OQL
# LDLIBS = -Wl,-E  -L$(O2HOME)/lib -lo2link -lo2sql -lo2 -lo2linkcuse
-lm -ldld
#Using O2Look and OQL
# LDLIBS = -Wl,-E  -L$(O2HOME)/lib -lo2lklink -lo2sql -lo2 -lo2linkcuse
-lm -ldld -lXm -lXt\ -lX11
#-------------------------------------------------------------#
# Here are the .o necessary to build the application:
# OBJS.appli = main.o Person.o  The user's application
#-------------------------------------------------------------#
OBJS.appli = main.o Person.o
.KEEP_STATE:
appli:$(OBJS.appli)
$(LINK.c) -v $(OBJS.appli) -o $@  $(LDLIBS)
```

Makefiles and Platforms

Refer to the documentation of your Operating system to write the Makefiles building your application.

In particular the **ld** command is different for each platform.

For O_2, depending on the platform you must use the following options for the link edition of your application:

- ***SUN-OS***

```
cc -Bstatic -o appli main.o Person.o -L$(O2HOME)/lib -lo2link -lo2\
-lo2linkcuse -lm
```

- ***HP UX***

```
cc -Wl, -E  -o appli main.o Person.o  -L$(O2HOME)/lib -lo2link -lo2\
-lo2linkcuse -lm -ldld
```

- ***IBM Aix***

```
cc -bM:SRE  -bE:$(O2HOME)/lib/o2.exp -o appli main.o Person.o\
-L$(O2HOME)/lib -lo2link -lo2  -lo2linkcuse -lm -lbsd
```

- ***IRIX***

```
cc -o appli main.o Person.o -L$(O2HOME)/lib -lo2link -lo2\
 -lo2linkcuse -lm -ldl
```

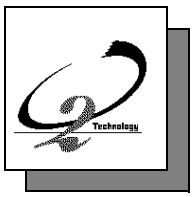
- ***Solaris***

```
/opt/SUNWspro/bin/cc -o appli main.o Person.o -L$(O2HOME)/lib\
-lo2link -lo2  -lm -lsocket -lnsl -lgen -ldl -lo2linkcuse
```

- ***NeXTStep***

```
cc -o appli main.o Person.o -L$(O2HOME)/lib -lo2link -lo2\ -all_load
```

3 An Example C Application



INDEX

Symbols

.c 52
.c file 52
.h file 43, 49

A

Application
C 13, 23
Code example 49
Example 43
Main Program 56
O_2C **16**, **37**
Run 16, **37**
Structure 15
Architecture
O_2 10

C

C 11
Application 13, 15, 43
Interface 14
Reading from 44
Writing from 44
C++
Interface 11
c_assign 16, **35**
Class definition 44
Collection 15
Iterators 16, **33**
Query 15
Commit 24
Connecting database 20
Converting handles 34

D

Database
Connection 20
Exit 39
Defining transactions 23
display 29

F

File
.c 52
.h 43
main.c 55
Function
c_assign 35
Call 15, 28
Iterator **33**
Manipulation 31
o2query 26, 27

G

Garbage Collection 38

H

Handle **15**, 29, 43
- Assignment 35
- Conversion 34
- Identifier 43
- Management **31**

I

Installation directory 21
Iterator **33**

J

Java 11

L

libname 21, 22
libo2link.a 20
libo2linkcuse.a 20
libpath 21
Libraries 20

M

Makefile **58**
Method
- Call 15, **28**
- **display** 29
- **read** 45
- **write** 46

Modules
- O_2 12

N

Named
- Object 16
- Value 16

O

O_2
- Advantages 13
- Architecture 10
- Collection 15
- Functions 15
- Integrated modules 12
- Library 20
- Methods 15

o2_assign_handle 31
o2_assign_name 32
o2_char 27
o2_garbage_base **38**
o2_handle 27
o2_integer 27

o2_iterator 33
o2_iterator_close 33
o2_iterator_next 33
o2_iterator_next_string 33
o2_iterator_open 33
o2_link_abort 15, 25
o2_link_commit 15, 24
o2_link_end 16, 39
o2_link_init 20–22
o2_link_set_base 22
o2_link_transaction 15, 23, 24
o2_link_validate 15, 24
o2_real 27
o2_run_application 16, 37
o2_string 27
o2_unref_handle 32
O_2C 11
O_2C
  Application **16**, **37**
  Function call 15, **28**
  Method call 15, 28
O_2Corba 11
O_2DBAccess 11
O_2Engine 10
O_2Graph 11
O_2Kit 11
O_2Look 11
O_2Look
  Libraries 20
O_2ODBC 11
o2query 26, 27
o2query_char 26
o2query_integer 26
o2query_real 26
o2query_string 27
O_2Store 10
O_2Tools 11
O_2Web 11
Object 31
  Assignment 16
  Identifier 43
OQL 11
  Function call 28
  Interface 26
  Method call 28
  Using in C 15

P

Persistence 13, 43
Platforms **58**
Pointer declarations 43
Program 23

Q

q_variable 27
Querying 15, **26–28**
querytext 27

R

read 46
Reading from C 44
Read-only
  Transactions 23
Run application **37**

S

set base 22
svname 21
swapdir 21
sysdir 21
sysname 21
System
    Architecture 10
    Features 13
systems file 21

T

Transaction **15**
    Definition 23
    Management **23–25**
    Read-only 23
    Update 24

U

Update transactions 24

V

Validate 25
Value 15, 31
    Assignment 16

W

write 46
Writing from C 44